CPSC 540 Assignment 2 (due February 6)

Large-Scale Machine Learning

0 Unnoffiial Course Evaluation

To help improve the course as we go along, or to suggest of how things could be done differently, please fill out the survey here:
`https://survey.ubc.ca/surveys/37-7974b781afc90962f53d98c2749/cpsc-540-informal-course-evaluation`

1 Convergence Rates

1.1 Gradient Descent

For minimizing a function f, in class we showed that if ∇f is Lipschitz continuous and f is strongly-convex then gradient descent iterations,

$$x^{t+1} = x^t - \alpha_t \nabla f(x^t),$$

with a step-size of $\alpha_t = 1/L$ satisfy

$$f(x^t) - f(x^*) = O(\rho^t),$$

for some $\rho < 1$ (we call this a linear convergence rate). In this question you'll show some related properties.

1. The rate above is in terms of the function value $f(x^t)$, but we might also be interested in the convergence of the iterates x^t to x^*. Show that if f is differentiable and strongly-convex then a convergence rate of $O(\rho^t)$ in terms of the function values implies that the iterations have a convergence rate of

$$\|x^t - x^*\| = O(\rho^{t/2}).$$

2. Consider using a constant step-size $\alpha_t = \alpha$ for some positive constant $\alpha < 2/L$. Show that gradient descent converges linearly under this alternate step-size (you can use the descent lemma).

3. In practice we typically don't L. A common strategy in this setting is to start with some small guess L^0 that we know is smaller than the true L (usually we take $L = 1$). On each iteration t, we initialize with $L^t = L^{t-1}$ and we check the inequality

$$f\left(x^t - \frac{1}{L^t}\nabla f(x^t)\right) \leq f(x^t) - \frac{1}{2L^t}\|\nabla f(x^t)\|^2.$$

   If this is not satisfied, we double L^t and test it again. This continues until we have an L^t satisfying the inequality. Show that gradient descent with $\alpha_t = 1/L^t$ defined in this way has a linear convergence rate of

$$f(x^t) - f(x^*) \leq \left(1 - \frac{\mu}{2L}\right)^t [f(x^0) - f(x^*)].$$

   Hint: if a function is L-Lipschitz continuous that it is also L'-Lipschitz continuous for any $L' \geq L$.

4. Describe a condition under which the step-sizes in the previous question would give a faster rate than $\rho = (1 - \mu/L)$.

Answers:

1. From strong-convexity and $\nabla f(x^*) = 0$ we have

$$\begin{aligned} f(x^t) &\geq f(x^*) + \nabla f(x^*)^T(x^t - x^*) + \frac{\mu}{2}\|x^t - x^*\|^2 \\ &= f(x^*) + \frac{\mu}{2}\|x^t - x^*\|^2. \end{aligned}$$

   Re-arranging we have

$$\frac{\mu}{2}\|x^t - x^*\|^2 \leq f(x^t) - f(x^*) = O(\rho^t).$$

   Taking the square root gives the result.

2. From the descent lemma we have

$$\begin{aligned} f(x^{t+1}) &\leq f(x^t) + \nabla f(x^t)(x^{t+1} - x^t) + \frac{L}{2}\|x^{t+1} - x^t\|^2 \\ &= f(x^t) - \alpha_t\|f(x^t)\|^2 + \frac{L\alpha_t^2}{2}\|\nabla f(x^t)\|^2 \\ &= f(x^t) - \frac{1}{2}\underbrace{\alpha_t\left(2 - L\alpha_t\right)}_{\gamma}\|\nabla f(x^t)\|^2. \end{aligned}$$

   Combining this with strong-convexity and subtracting $f(x^*)$ from both sides we get

$$f(x^{t+1}) - f(x^*) \leq (1 - \mu\gamma)\left[f(x^t) - f(x^*)\right].$$

   Applying this recursively and using that $\alpha_t < 2/L$ implies $\gamma > 0$ gives the result (we get our previous result in the special case of $\alpha_t = 1/L$ which implies $\gamma = 1/L$).

3. First we note that the inequality must be satisfied for $L^t > L$ by L-Lipschitz continuity of the gradient (an L-Lipschitz continuous function is also M-Lipschitz for any $M > L$). This means we have $L^t \leq 2L$, since once this is satisfied we won't double L again. Because of the way we set L^t, for every iteration t we have

$$\begin{aligned} f(x^{t+1}) &\leq f(x^t) - \frac{1}{2L^t}\|\nabla f(x^t)\|^2 \\ &\leq f(x^t) - \frac{1}{4L}\|\nabla f(x^t)\|^2. \end{aligned}$$

   The remaining steps follow as before.

4. Any condition that guarantees $L^t < L$ for all t will work. For example, if the iterates x^t all stay within a ball around x^* where $L^t << L$ then this approach would be substantially faster.

1.2 Sign-Based Gradient Descent

In some situations it might be hard to accurately compute the elements of the gradient, but we might have access to the sign of the gradient. Consider an f that is strongly-convex and where ∇f is Lipschitz continuous in the ∞-norm, meaning that

$$f(y) \leq f(x) + \nabla f(x)^T(y - x) + \frac{L_\infty}{2}\|y - x\|_\infty^2,$$

for all y and x and some L_∞.

For this setting, consider a sign-based gradient descent algorithm of the form

$$x^{t+1} = x^t - \frac{\|\nabla f(x^t)\|_1}{L_\infty}\text{sign}(\nabla f(x^t)),$$

where we define the sign function element-wise as

$$\text{sign}(x_j) = \begin{cases} +1 & x_j > 0 \\ 0 & x_j = 0 \\ -1 & x_j < 0 \end{cases},$$

1. Show that the sign-based gradient descent method satisfies

$$f(x^{t+1}) - f(x^*) \leq \left(1 - \frac{\mu}{L_\infty}\right)[f(x^t) - f(x^*)].$$

2. To compare this rate to the rate of gradient descent, we need to know the relationship between the usual Lipschitz constant L (in the 2-norm) and L_∞. Show that the relationship between these constants is

$$L \leq L_\infty \leq dL,$$

where L_∞ and L are the minimal values of the Lipschitz constants.
Hint: you may want to review the notes on norms and the notes on convexity inequalities from the course webpage.

Answers:

1. Plugging in the definition of the iteration we have

$$\begin{aligned} f(x^{t+1}) &\leq f(x^t) + \nabla f(x^t)^T(x^{t+1} - x^t) + \frac{L}{2}\|x^{t+1} - x^t\|_\infty^2 \\ &= f(x^t) - \frac{\|\nabla f(x^t)\|_1}{L}\nabla f(x^t)^T\text{sign}(\nabla f(x^t)) + \frac{\|\nabla f(x^t)\|_1^2}{2L}\|\text{sign}(\nabla f(x^t))\|_\infty^2 \\ &= f(x^t) - \frac{1}{L}\|\nabla f(x^t)\|_1^2 + \frac{1}{2L}\|\nabla f(x^t)\|_1^2 \\ &= f(x^t) - \frac{1}{2L}\|\nabla f(x^t)\|_1^2 \\ &\leq f(x^t) - \frac{1}{2L}\|\nabla f(x^t)\|^2, \end{aligned}$$

where the last line uses that $\|v\|_1 \geq \|v\|$. It follows that

$$f(x^{t+1}) - f(x^*) \leq \left(1 - \frac{\mu}{L}\right)[f(x^t) - f(x^*)].$$

2. We have

$$\begin{aligned} f(y) &\leq f(x) + \nabla f(x)^T(y - x) + \frac{L_\infty}{2}\|y - x\|_\infty^2 \\ &\leq f(x) + \nabla f(x)^T(y - x) + \frac{L_\infty}{2}\|y - x\|^2, \end{aligned}$$

since $\|v\|_\infty \leq \|v\|$. This means that if ∇f is L_∞-Lipschitz in the $\infty-$-norm then it must be L_∞-Lipschitz in the 2-norm. But this leaves open the possibility that L is smaller than L_∞ so we have $L \leq L_\infty$. We similarly have

$$\begin{aligned} f(y) &\leq f(x) + \nabla f(x)^T(y-x) + \frac{L}{2}\|y-x\|^2 \\ &\leq f(x) + \nabla f(x)^T(y-x) + \frac{dL}{2}\|y-x\|_\infty^2, \end{aligned}$$

since $\|v\| \leq \sqrt{d}\|v\|_\infty$. This means that if ∇f is L-Lipschitz in the 2-norm then it must be dL-Lipschitz in the ∞-norm. But this leaves open the possibility that L_∞ is smaller than dL, so we have $L_\infty \leq dL$.

1.3 Block Coordinate Descent

Consider a problem it makes sense to partition our variables into k disjoint 'blocks'

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \end{bmatrix},$$

each of size d/k. Assume that f is *strongly-convex*, and *blockwise strongly-smooth*,

$$\nabla^2 f(w) \succeq \mu I, \quad \nabla^2_{bb} f(w) \preceq LI,$$

for all w and all blocks b. Consider a *block coordinate descent* algorithm where we use the iteration

$$x^{t+1} = x^t - \frac{1}{L}(\nabla f(x^t) \circ e_{b_t}),$$

where b_t is the block we choose on iteration t, e_b is vector of zeros with ones at the locations of block b, and $\circ$ means element-wise multiplication of the two vectors. (It's like coordinate descent except we're updating d/k variables instead of just one.)

1. Assume that we pick a random block on each iteration, $p(b_t = b) = 1/k$. Show that this method satisfies
$$\mathbb{E}[f(x^{t+1})] - f(x^*) \leq \left(1 - \frac{\mu}{Lk}\right)[f(x^t) - f(x^*)].$$

2. Assume that each block b has its own strong-smoothness constant L_b,
$$\nabla^2_{bb} f(w) \preceq L_b I,$$
so that the strong-smoothness constant from part 1 is given by $L = \max_b\{L_b\}$. Show that if we sample the blocks proportional to L_b, $p(b_t = b) = \frac{L_b}{\sum_{b'} L_{b'}}$ and we use a larger step-size of $1/L_{b_t}$, then we obtain a faster convergence rate provided that some $L_b \neq L$.

Answer:

1.

$$\begin{aligned} f(x^{t+1}) &= f(x^t) + \nabla_{b_t} f(x^t)^T (x^{t+1} - x^t)_{b_t} + \frac{1}{2}(x^{t+1} - x^t)_{b_t}^T \nabla_{bb}^2 f(z)(x^{t+1} - x^t)_{b_t} \\ &\leq f(x^t) + \nabla_{b_t} f(x^t)^T (x^{t+1} - x^t)_{b_t} + \frac{L}{2}\|(x^{t+1} - x^t)_{b_t}\|^2 \\ &= f(x^t) - \frac{1}{L}\|\nabla_{b_t} f(x^t)\|^2 + \frac{L}{2}\|\frac{1}{L}\nabla_{b_t} f(x^t)\|^2 \\ &= f(x^t) - \frac{1}{2L}\|\nabla_{b_t} f(x^t)\|^2. \end{aligned}$$

This holds for any block b_t, but using random selection we have

$$\begin{aligned} \mathbb{E}[f(x^{t+1})] &\leq f(x^t) - \frac{1}{2L}\sum_b \frac{1}{k}\|\nabla_b f(x^t)\|^2 \\ &= f(x^t) - \frac{1}{2kL}\|\nabla f(x^t)\|^2, \end{aligned}$$

and by strong-convexity we have

$$\mathbb{E}[f(x^{t+1}) - f(x^*)] \leq \left(1 - \frac{\mu}{kL}\right)[f(x^t) - f(x^*)].$$

Applying this recursively gives the result.

2. From the reasoning above applied but using L_{b_t} to bound the blockwise Taylor expansion we have

$$f(x^{t+1}) \leq f(x^t) - \frac{1}{2L_{b_t}}\|\nabla_{b_t} f(x^t)\|^2.$$

Taking the expectation we now have

$$\begin{aligned} \mathbb{E}[f(x^{t+1})] &\leq f(x^t) - \frac{1}{2}\sum_b \frac{1}{L_b}\frac{L_b}{\sum_{b'} L_{b'}}\|\nabla_b f(x^t)\|^2 \\ &= f(x^t) - \frac{1}{2\sum_{b'} L_{b'}}\|\nabla f(x^t)\|^2. \end{aligned}$$

Using strong-convexity we get

$$\mathbb{E}[f(x^{t+1}) - f(x^*)] \leq \left(1 - \frac{\mu}{\sum_{b'} L_{b'}}\right)[f(x^t) - f(x^*)].$$

We always have $\sum_{b'} L_{b'} \leq \sum_{b'} L = Lk$, so if some $L_b < L$ this inequality is strict and the result above is faster.

2 Large-Scale Algorithms

2.1 Coordinate Optimization

The function *example_logistic* loads a dataset and tries to fit an L2-regularized logistic regression model using coordinate optimization. Unfortunately, if we use L_f as the Lipschitz constant of ∇f, the runtime of this procedure is $O(d^3 + nd^2\frac{L_f}{\mu}\log(1/\epsilon))$. This comes from spending $O(d^3)$ computing L_f, having an iteration cost of $O(nd)$, and requiring a $O(d\frac{L_f}{\mu}\log(1/\epsilon))$ iterations. This non-ideal runtime is also reflected in practice: the algorithm's iterations are relatively slow and even after 500 "passes" through the data it isn't particularly close to the optimal function value.

1. Modify this code so that the runtime of the algorithm is $O(nd\frac{L_c}{\mu}\log(1/\epsilon))$, where L_c is the Lipschitz constant of *all* partial derivatives $\nabla_i f$. You can do this by modifying the iterations so they have a cost $O(n)$ instead of $O(nd)$, and instead of using a step-size of $1/L_f$ they use a step-size of $1/L_c$ (which is given by $\frac{1}{4}\max_j\{\|x_j\|^2\} + \lambda$). Hand in your code and report the final function value and total time.

2. To further improve the performance, make a new version of the code which samples the variable to update j_t proportional to the individual Lipschitz constants L_j of the coordinates, and use a step-size of $1/L_{j_t}$. You can use the function *sampleDiscrete* to sample from discrete distribution given the probability mass function. Hand in your code, and report the final function value as well as the number of passes.

3. Report the number of passes the algorithm takes as well as the final function value if you use *uniform sampling* but use a step-size of $1/L_{j_t}$.

4. Suppose that when we use a step-size of $1/L_{j_t}$, we see that uniform sampling outperforms Lipschitz sampling. Why would this be consistent with the bounds we stated in class?

Answers:

1. The code should look roughly like this (note the different calculation of L, we update Xw rather than re-compute it, and we only compute one element of the gradient).

```
function [model] = logisticL2(X,y,lambda)

% Add bias variable
[n,d] = size(X);
X = [ones(n,1) X];
d = d+1;

% Initial values of regression parameters
w = zeros(d,1);

% Optimizaion parameters
maxPasses = 500;
progTol = 1e-4;
L = .25*max(sum(X.^2)) + lambda;

w_old = w;
Xw = zeros(n,1);
for i = 1:maxPasses*d

    % Choose variable to update 'j'
    j = randi(d);

    % Compute partial derivative 'g_j'
    sigmoid = 1./(1+exp(-y.*Xw));
    g_j = -X(:,j)'*(y.*(1-sigmoid)) + lambda*w(j);

    % Update variable
    wj_old = w(j);
    w(j) = w(j) - (1/L)*g_j;
    Xw = Xw + X(:,j)*(w(j)-wj_old);

    % Check for lack of progress after each "pass"
    if mod(i,d) == 0
```

```
        change = norm(w-w_old,inf);
        fprintf('Passes = %d, function = %.4e, change = %.4f\n',i/d,logisticL2_loss(w,
        if change < progTol
            fprintf('Parameters changed by less than progTol on pass\n');
            break;
        end
        w_old = w;
    end
end


model.w = w;
model.predict = @predict;
end

function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);
Xhat = [ones(t,1) Xhat];
w = model.w;
yhat = sign(Xhat*w);
end
```

The performance will vary based on the randomization/hardware, but on my laptop this decreased the time from around 20 seconds down to around 4 seconds. The algorithm still takes 500 iterations and finds a slightly lower function value of $1.473e+02$ in Matlab notation.

2. The code should look roughly like this (note that we now compute a vector of L values, use *sampleDiscrete* to pick j, and use $L(j)$ as the step-size).

```
function [model] = logisticL2(X,y,lambda)

% Add bias variable
[n,d] = size(X);
X = [ones(n,1) X];
d = d+1;

% Initial values of regression parameters
w = zeros(d,1);

% Optimizaion parameters
maxPasses = 500;
progTol = 1e-4;
L = .25*sum(X.^2) + lambda;

w_old = w;
Xw = zeros(n,1);
for i = 1:maxPasses*d

    % Choose variable to update 'j'
    j = sampleDiscrete(L/sum(L));

    % Compute partial derivative 'g_j'
    sigmoid = 1./(1+exp(-y.*Xw));
```

```
        g_j = -X(:,j)'*(y.*(1-sigmoid)) + lambda*w(j);

        % Update variable
        wj_old = w(j);
        w(j) = w(j) - (1/L(j))*g_j;
        Xw = Xw + X(:,j)*(w(j)-wj_old);

        % Check for lack of progress after each "pass"
        if mod(i,d) == 0
            change = norm(w-w_old,inf);
            fprintf('Passes = %d, function = %.3e, change = %.3f\n',i/d,logisticL2_loss(w,
            if change < progTol
                fprintf('Parameters changed by less than progTol on pass\n');
                break;
            end
            w_old = w;
        end
    end


    model.w = w;
    model.predict = @predict;
    end

    function [yhat] = predict(model,Xhat)
    [t,d] = size(Xhat);
    Xhat = [ones(t,1) Xhat];
    w = model.w;
    yhat = sign(Xhat*w);
    end
```

Technically, using *sampleDiscrete* costs $O(d)$ so we've increased our iteration cost to $O(n+d)$, but for an initial $O(d)$ cost of computing the CDF we could implement the iterations in $O(n+\log(d))$. This approach gives a lower function value, achieving something like $1.436e+02$ and tending to terminate early rather than running for 500 passes.

3. This strategy reduces the function value to $1.409e+02$ and the method terminates after around 150 passes (which takes about 1 second on my laptop).

4. When we analyzed uniform sampling we used a step-size of $1/L_c$ and when we analyzed Lipschitz sampling we said the we use a step-size of $1/L_{j_t}$. In the above case we're using a step-size of $1/L_{j_t}$ with uniform sampling, which should give a tighter bound than using L_c since it makes more progress per iteration.

2.2 Proximal-Gradient

If you run the demo *example_group*, it will load a dataset and fit a multi-class logistic regression (softmax) classifier. This dataset is actually *linearly-separable*, so there exists a set of weights W that can perfectly classify the training data (though it may be difficult to find a W that perfectly classifiers the validation data). However, 90% of the columns of X are irrelevant. Because of this issue, when you run the demo you find that the training error is 0 while the test error is something like 0.2980.

1. Write a new function, *softmaxClassifierL2*, that fits a multi-class logistic regression model with L2-regularization (this only involves modifying the objective function). Hand in the modified loss function and report the best validation error achievable with $\lambda = 10$ (which is best value among powers to 10). Also report the number of non-zero parameters in the model and the number of original features that the model uses.

2. While L2-regularization reduces overfitting a bit, it still uses all the variables even though 90% of them are irrelevant. In situations like this, L1-regularization may be more suitable. Write a new function, *softmaxClassifierL1*, that fits a multi-class logistic regression model with L1-regularization. You can use the function *proxGradL1*, which minimizes the sum of a differentiable function and an L1-regularization term. Report the number of non-zero parameters in the model and the number of original features that the model uses.

3. L1-regularization achieves sparsity in the *model parameters*, but in this dataset it's actually the *original features* that are irrelevant. We can encourage sparsity in the original features by using *group* L1-regularization. Write a new function, *proxGradGroupL1*, to allow (disjoint) *group* L1-regularization. Use this within a new function, *softmaxClassiferGL1*, to fit a group L1-regularized multi-class logistic regression model (where *rows* of W are grouped together and we use the L2-norm of the groups). Hand in both modified functions (*softmaxClassifierGL1* and *proxGradGroupL1*) and report the validation error achieved with $\lambda = 10$. Also report the number of non-zero parameters in the model and the number of original features that the model uses.

Answer:

1.

The updated loss function should look roughly like this:

```
function [model] = softmaxClassifierL2(X,y,lambda)

% Compute sizes
[n,d] = size(X);
k = max(y);

W = zeros(d,k); % Each column is a classifier
W(:) = findMin(@softmaxLoss,W(:),500,1,X,y,k,lambda);

model.W = W;
model.predict = @predict;
end

function [yhat] = predict(model,X)
W = model.W;
[~,yhat] = max(X*W,[],2);
end

function [nll,g,H] = softmaxLoss(w,X,y,k,lambda)

[n,d] = size(X);
W = reshape(w,[d k]);

XW = X*W;
Z = sum(exp(XW),2);
```

```
ind = sub2ind([n k],[1:n]',y);
nll = -sum(XW(ind)-log(Z)) + (lambda/2)*sum(W(:).^2);

g = zeros(d,k);
for c = 1:k
    g(:,c) = X'*(exp(XW(:,c))./Z-(y == c));
end
g = reshape(g,[d*k 1]) + lambda*W(:);
end
```

If $\lambda = 10$, then the error decreases from 0.2990 down to 0.2740. All 500 model parameters are non-zero, and all 100 original features are used.

2.

If $\lambda = 10$, then the error decreases to 0.0840. With this choice, there are only 35 non-zero model parameters and only 19 of the original features are used (the exact values will again vary depending on the implementation and the solution is not unique).

3.

A generic way to implement groups is to use a vector *groups* that contains the group number of each variable. With this data structure, you could modify *softmaxClassifier* to call:

```
funObj = @(W)softmaxLoss(W(:),X,y,k);
groups = repmat([1:d]',1,k);
W(:) = proxGradGL1(funObj,W(:),lambda,500,groups);
```

In *proxGradL1*, the only thing you need to change is the proximal operator (though you should also change the calculation of the function in the *fprintf* statement if you want it to be accurate):

```
function [w] = groupSoftThreshold(w,threshold,groups)
    nGroups = max(groups);
    reg = sqrt(accumarray(groups(groups~=0),w(groups~=0).^2));
    for g = 1:nGroups
        if reg(g) ~= 0
            w(groups==g) = (w(groups==g)/reg(g))*max(0,reg(g)-threshold);
        end
    end
end
```

If $\lambda = 10$, then the error decreases to 0.0540. The number of non-zero parmaeters increases to 115, but these are only spread across 23 original features (the exact values will again vary depending on the implementation and the solution is not unique).

2.3 Stochastic Gradient

If you run the demo *example_stochastic*, it will load a dataset and try to fit an L2-regularized logistic regression model using 10 "passes" of stochastic gradient using the step-size of $\alpha_t = 1/\lambda t$ that is suggested in many theory papers. Note that the demo is quite slow as Matlab doesn't do well with 'for' loops, but if you implemented this in C this would be very fast even though there are 50,000 training examples.

Unfortunately, even if we ignore the Matlab-slowness, the performance of this stochastic gradient method is atrocious. It often goes to areas of the parameter space with the objective function overflows and the final value is usually in the range of something like $6.5 - 7.5 \times 10^4$. This is quite far from the solution of 2.7068×10^4 and is even worse than just choosing $w = 0$ which gives 3.5×10^4. (This is unlike gradient descent and coordinate optimization, which never increase the objective function.)

1. Although $\alpha_t = 1/\lambda$ gives the best possible convergence rate in the worst case, in practice it's typically horrible (as we're not usually opitmizing the hardest possible λ-strongly convex function). Experiment

with different choices of step-size to see if you can get better performance. Report the step-size that you found gave the best performance, and the objective function value obtained by this strategy for one run.

2. Besides tuning the step-size, another strategy that often improves the performance is using a (possibly-weighted) average of the iterations w^t. Explore whether this strategy can improve performance. Report the performance with an averaging strategy, and the objective function value obtained by this strategy for one run. (Note that the best step-size sequence with averaging might be different than without averaging.)

3. A popular variation on stochastic is AdaGrad, which uses the iteration

$$w^{t+1} = w^t - \alpha_t D_t \nabla f(w^t),$$

where the element in position (j, j) of the diagonal matrix D_t is given by $1/\sqrt{\delta + \sum_{k=0}^{t}(\nabla_j f_{i_k}(w^k))^2}$. Here, i_k is the example i selected on iteration k and ∇_j denotes element j of the gradient (and in AdaGrad we typically don't average the steps). Implement this algorithm and experiment with the tuning parameters α_t and δ. Hand in your code as well as the best step-size sequence you found and again report the performance for one run.

4. Impelement the SAG algorithm with a step-size of $1/L$, where the L is the maximum Lipschitz constant across the training examples ($L = 0.25 \max_i\{\|x^i\|^2\} + \lambda$). Hand in your code and again report the performance for one run.

Answers:

1. I found that you could get much better performance by just setting $\alpha_t = 10^{-4}$, and on one run this obtained 2.7124×10^4.

2. By averaging all the iterations I found I could get better performance by setting $\alpha_t = 10^{-3}$, and on one run this obtained 2.7073×10^4.

3. I found using $\delta = 1$ and $\alpha_t = 0.1$ worked ok, and on one run got 2.7125×10^4 (e.g., it didn't really outperform averaging or just using a constant step-size). The code could look like this:

```
w = zeros(d,1);
w_old = w;
D = ones(d,1);
for t = 1:maxPasses*n

    % Choose variable to update
    i = randi(n);

    % Evaluate the gradient for example i
    [f,g] = logisticL2_loss(w,X(i,:),y(i),lambda);

    % Choose the step-size
    alpha = .1;

    % Update the diagonal scaling
    D = D + g.^2;

    % Take the stochastic gradient step
    w = w - alpha*g./sqrt(D);

    if mod(t,n) == 0
        change = norm(w-w_old,inf);
        fprintf('Passes = %d, function = %.4e, change = %.4f\n',t/n,logisticL2_loss(w,X,y,lambdaFull),change);
        if change < progTol
            fprintf('Parameters changed by less than progTol on pass\n');
            break;
        end
        w_old = w;
    end
end
```

4. The code could look like this:

```
w = zeros(d,1);
w_old = w;
G = zeros(n,d);
D = zeros(d,1);
for t = 1:maxPasses*n

    % Choose variable to update
    i = randi(n);

    % Evaluate the gradient for example i
    [f,g] = logisticL2_loss(w,X(i,:),y(i),lambda);

    % Choose the step-size
    alpha = 1/L;

    % Update the direction
    D = D + g - G(i,:)';
    G(i,:) = g;

    % Take the stochastic gradient step
    w = w - alpha*D/n;

    if mod(t,n) == 0
        change = norm(w-w_old,inf);
        fprintf('Passes = %d, function = %.4e, change = %.4f\n',t/n,logisticL2_loss(w,X,y,lambdaFull),change);
        if change < progTol
            fprintf('Parameters changed by less than progTol on pass\n');
            break;
        end
        w_old = w;
    end
end
```

I found that SAG tended to find the solution up to five significant digits, 2.7068×10^4, after 10 passes.

3 Kernels and Duality

3.1 Fenchel Duality

Recall that the Fenchel dual for the primal problem

$$P(w) = f(Xw) + g(w),$$

is the dual problem

$$D(z) = -f^*(-z) - g^*(X^T z),$$

or if we re-parameterize in terms of $-z$:

$$D(z) = -f^*(z) - g^*(-X^T z), \tag{1}$$

where f^* and g^* are the convex conjugates. Convex conjugates are discussed in Section 3.3 of Boyd and Vandenberghe (`http://stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf`). Read this, then derive the Fenchel dual for the following problems:

1. $P(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{\lambda}{2}\|w\|^2$ (L2-regularized least squares)
2. $P(w) = \|Xw - y\|_1 + \lambda\|w\|_1$ (robust regression with L1-regularization)
3. $P(w) = \sum_{i=1}^{N} \log(1 + \exp(-y^i w^T x^i)) + \frac{\lambda}{2}\|w\|^2$ (regularized maximum entropy)

Hint: Don't try to take the Lagrangian dual, a generic strategy to compute the special case of Fenchel duals is as follows:

- Determine X, f, and g to put the problem into the primal format.
- Determine the form of f^* and g^* (note that A here is not relevant).
- Evaluate f^* at $-z$ and g^* at $X^T z$ to get the final form.

For a differentiable f, you can often solve for the value achieving the sup inside of $f^*(v)$ by taking the gradient of $(x^T v - f(x))$ and setting it to zero (keeping in mind whether there are values of v where the sup might be infinity). Example 3.26 in the book gives the convex conjugate in the case where f is a norm. Section 3.3.2 of the book shows how the convex conjugate changes if you scale a function and/or compose a function with an affine transformation. For parts 1 and 2, the X in the primal will just be the data matrix X. But for part 3, it will be easier if you define X as a matrix with row i is given by $y^i x^i$. For part 3 you'll want to use $f(v) = \sum_{i=1}^{n} \log(1 + \exp(v_i))$, which is a *separable* function (meaning that you can optimize each z_i independently).

Answer:

1. We can use
$$f(x) = \frac{1}{2}\|x - y\|^2, \quad g(w) = \frac{\lambda}{2}\|w\|^2.$$
To put the problem into the primal form. We can compute the convex conjugate of f,
$$f^*(z) = \sup_x \{x^T z - \frac{1}{2}\|x - y\|^2\},$$
by taking the derivative of the function inside the sup and setting it to zero,
$$0 = z - (x - y), \text{ or } x = z + y.$$
Plugging this value back in we get
$$f^*(z) = (z + y)^T z - \frac{1}{2}\|(z + y) - y\|^2 = z^T z + y^T z - \frac{1}{2} z^T z = \frac{1}{2}\|z\|^2 + y^T z.$$
We can use the scaling rule above and the squared 2-norm result from class to get
$$g^*(z) = \frac{\lambda}{2}\|\frac{1}{\lambda} z\|^2 = \frac{1}{2\lambda}\|z\|^2.$$
Using these we get a dual of
$$D(z) = -\frac{1}{2}\|z\|^2 + z^T y - \frac{1}{2\lambda} z^T X X^T z,$$
or equivalently
$$D(z) = -\frac{1}{2}\|z\|^2 - z^T y - \frac{1}{2\lambda} z^T X X^T z.$$
It's seems weird that these two problems are both dual to the original problem since one of the terms has a different sign. But notice that if z solves the first one then $-z$ solves the second one (the first and third term don't depend on the sign of z). Notice that we can use the kernel trick in this formulation to replace XX^T.

2. We can use
$$f(x) = \frac{1}{2}\|x - y\|_1, \quad g(w) = \lambda\|w\|_1,$$
The dual norm of the L1-norm is the ℓ_∞-norm, so Example 3.26 shows that if $h(z) = \|z\|_1$ then the convex conjugate is the ℓ_∞-norm unit ball. But in our case the L1-norm is multiplied by λ, so using that the conjugate of $h(x) = \alpha f(x)$ is $h^*(z) = \alpha f^*((1/\alpha)z)$ we get
$$g^*(z) = \begin{cases} 0 & \|z\|_\infty \leq \lambda \\ \infty & \text{otherwise} \end{cases}.$$

For f, we use the result for a scaled 1-norm and the rule for composing with an affine transformation, that if $h(x) = f(x - y)$ then $h^*(z) = f^*(z) + y^T z$. This gives

$$f^*(z) = \begin{cases} y^T z & \|z\|_\infty \leq 1 \\ \infty & \text{otherwise} \end{cases}.$$

Plugging this into the form of the dual we get a dual of

$$D(z) = y^T z \text{ subject to } \|z\|_\infty \leq 1, \|X^T z\|_\infty \leq \lambda,$$

and again notice that the sign of z doesn't matter. (Notice that this is a linear program.)

3. Let's define X as a matrix with row i is given by $y^i x^i$. Using this, we can use

$$f(x) = \sum_{i=1}^{n} \log(1 + \exp(z_i)), \quad g(w) = \frac{\lambda}{2}\|w\|^2.$$

We addressed this particular g above, so we just need to figure out f^*. First, we notice that f^* is *separable* so we have

$$\begin{aligned} f^*(z) &= \sup_x\{z^T x - f(x)\} \\ &= \sup_x\{z^T x - \sum_{i=1}^{n} \log(1 + \exp(z))\} \\ &= \sum_{i=1}^{n} \sup_x\{z_i x - \log(1 + \exp(z_i))\} \end{aligned}$$

So we can get the conjugate if we can solve the single-variable problem

$$h^*(z) = \sup_x\{zx - \log(1 + \exp(z))\}.$$

Taking the derivative and setting it to zero we get

$$\begin{aligned} 0 &= z - \frac{\exp(x)}{1 + \exp(x)} \\ 0 &= z - \frac{1}{1 + \exp(-x)} \\ 0 &= z(1 + \exp(-x)) - 1 \\ 1 - z &= z\exp(-x) \\ \log(1 - z) &= \log(z) - x \\ x &= \log\left(\frac{z}{1 - z}\right). \end{aligned}$$

which only exists if $0 < z < 1$. Plugging this into h^* for $0 < z < 1$ we get

$$\begin{aligned} h^*(z) &= z\log\left(\frac{z}{1 - z}\right) - \log(1 + \exp(\log\left(\frac{z}{1 - z}\right))) \\ &= z\log z - z\log(1 - z) - \log(1 + \frac{z}{1 - z}) \\ &= z\log z - z\log(1 - z) - \log(\frac{1}{1 - z}) \\ &= z\log z + (1 - z)\log(1 - z) \\ &= -H(y), \end{aligned}$$

where is the binary negative entropy (notice that if $y > 1$ or $y < 0$ then we have $H(h) = \infty$). Putting everything together we get a dual of

$$D(z) = \sum_{i=1}^{n} H(-z) - \frac{1}{2\lambda}\|X^T z\|^2, \text{ subject to } -1 \leq z \leq 0,$$

or

$$D(z) = \sum_{i=1}^{n} H(z) - \frac{1}{2\lambda}\|X^T z\|^2, \text{ subject to } 0 \leq z \leq 1.$$

Thus, the dual problem can be viewed maximizing entropy subject to regularization based on the data matrix X.

3.2 Stochastic Dual Coordinate Ascent

The dual of the SVM problem,

$$P(w) = \sum_{i=1}^{N} \max\{0, 1 - y^i w^T x^i\} + \frac{\lambda}{2}\|w\|^2,$$

is

$$D(z) = e^T z - \frac{1}{2\lambda} z^T Y X X^T Y z, \quad \text{s.t. } 0 \leq z_i \leq 1, \forall_i.$$

where e is a vector of ones, Y is diagonal matrix with the y^i values along the diagonal, and we have $w^* = \frac{1}{\lambda} X^T Y z^*$. Starting from *example_dual.m*, implement a dual coordinate optimization strategy to optimize the SVM objective. Hand in your code, report the optimal value of the primal and dual objectives with $\lambda = 1$, and report the number of support vectors

Hint: the objective function is a quadratic and the constraints are just lower and upper bounds. This lets you derive the optimal update for one variable with the other held fixed: solve for the value of z_i with a partial derivative of zero, and if this violates the constraints then the solution must be either $z_i = 0$ or $z_i = 1$ (depending on which one is lower).

Answer:

Below is the exact for coordinate j, and the optimal dual value is 90.68.

```
i = ceil(rand*n); % Uniform sampling
z(i) = (lambda - z'*G(:,i) + z(i)*G(i,i))/G(i,i);
z(i) = min(1,max(z(i),0));
```

The assignment doesn't ask for a termination criterion, but a principled one is to occasionally compute the primal and dual and check whether the difference (called the *duality gap*) is sufficiently small.

```
% Check duality gap to decide when to stop
D = sum(z) - (z'*G*z)/(2*lambda);
w = (1/lambda)*(yX'*z);
P = sum(max(1-y.*(X*w),0)) + (lambda/2)*(w'*w);
gap = P-D
if gap < 1e-3
    break;
end
```

The number is support vectors is approximately 101 out the possible 270.

3.3 Large-Scale Kernel Methods

The function *kernelRegression.m* implements kernel regression with the squared error, L2-regularizer, and Gaussian kernel. If you run your cross-validation code from Assignment 1 Question 1.2, you'll find that it achieves similar performance to using Gaussian RBFs.

1. Report the λ and σ reported using cross-validation on this previous assignment question. What are the (approximate) relationships between λ and σ between the two models (the one with Gaussian RBFs and the other with Gaussian kernels).
2. Implement the *subset of regressors* model for large-scale kernel methods we discussed in class. Hand in your code and report the qualitative performance (describe how well the model fits the data visually) for small and large values of the number of regressors m.
3. Implement the *random kitchen sink* model for large-scale kernel methods we discussed in class. Hand in your code and contrast the performance of this method with the subset of regressors model, for both large and small m.

Answer:

1. The best value of σ is again roughly $\frac{1}{2}$, so it seems to be the same between the two models. The best value of λ is now approximately 10^{-3}, which is approximately the square root of the λ we obtained for Gaussian RBFs. (This makes sense because Gaussian RBFs use K^TK, so the eigenvalues and condition number get squared.)
2. With a small m, I found that the subset of regressors method gave results that fit the overall trend of the function but missed the oscillations. As m increased, it converged to the solution obtained when we use the full dataset. Code implementing the method could look like this

```
function [model] = kernelRegression_nystrom(X,y,lambda,sigma)

% Compute sizes
[n,d] = size(X);

m = 25;
Xsub = X(1:m,:);
Kmn = rbfBasis(Xsub,X,sigma);
Kmm = rbfBasis(Xsub,Xsub,sigma);
z = (Kmn*Kmn' + lambda*Kmm)\(Kmn*y);

model.X = X;
model.Xsub = Xsub;
model.z = z;
model.sigma = sigma;
model.predict = @predict;

end

function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);

Ktm = rbfBasis(Xhat,model.Xsub,model.sigma);

yhat = Ktm*model.z;
end
```

```
function [Xrbf] = rbfBasis(X1,X2,sigma)
n1 = size(X1,1);
n2 = size(X2,1);
d = size(X1,2);
Z = 1/sqrt(2*pi*sigma^2);
D = X1.^2*ones(d,n2) + ones(n1,d)*(X2').^2 - 2*X1*X2';
Xrbf = Z*exp(-D/(2*sigma^2));
end
```

3. Code implementing the method could look like this

```
function [model] = kernelRegression_kitchen(X,y,lambda,sigma)

% Compute sizes
[n,d] = size(X);


% Training
m = 25;
R = randn(d,m);
R = normrnd(zeros(d,m),sigma*ones(d,m));
Z = exp(i*X*R);
w = (Z'*Z + lambda*eye(m))\(Z'*y);

model.w = w;
model.R = R;
model.sigma = sigma;
model.predict = @predict;
end

function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);

yhat = real(exp(i*Xhat*model.R)*model.w);
end

function [Xrbf] = rbfBasis(X1,X2,sigma)
n1 = size(X1,1);
n2 = size(X2,1);
d = size(X1,2);
Z = 1/sqrt(2*pi*sigma^2);
D = X1.^2*ones(d,n2) + ones(n1,d)*(X2').^2 - 2*X1*X2';
Xrbf = Z*exp(-D/(2*sigma^2));
end
```

I found that it worked pretty similarly to the subset of regressors method.